Operating Systems (2INC0)

Deadlock (08)
Terminology

Dr. Geoffrey Nelissen

Courtesy of Prof. Dr. Johan Lukkien and
Dr. Tanir Ozcelebi
(also thanks to Bic & Shaw, Silberschatz, Galvin & Gagne)

IRiS Interconnected Resource-aware Intelligent Systems

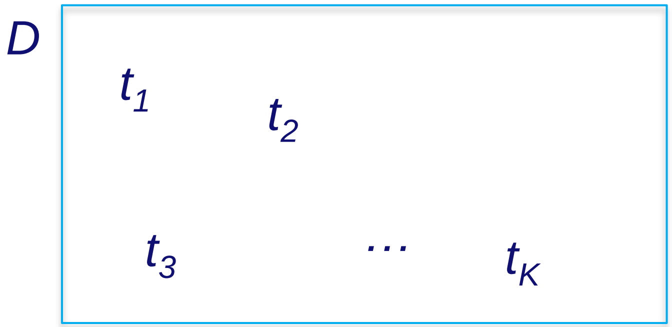
TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Formal definitions

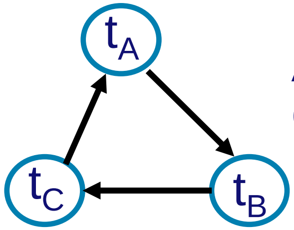
- We call a task ***blocked*** if:
  - it is **waiting on a blocking synchronization action**

- A set D of tasks is called ***deadlocked*** if
  - **all tasks in *D* are blocked or terminated** (normally or abnormally),
  - there is **at least one non-terminated** task in D, and
  - for each non-terminated task t in D, any **task that might unblock *t* is also in *D*.**



Deadlock: conditions

- Program behaviors that may lead to deadlock
  - mutual exclusion
  - greediness: hold and wait
    - incrementally reserving some resources while waiting for other resources to become available (e.g., dining philosophers).
  - absence of preemption mechanism
  - circular waiting
    - e.g., tasks t_A, t_B and t_C waiting on each other:



A *wait-for* graph (explained next).

- These all play a role and can be (should be) addressed explicitly in the solution.
  - i.e., deadlock is addressed by *avoiding / working better with* these behaviors

Deadlock: type of resources

- Deadlock is usually associated with access to resources.

  - ***consumable* resources**: resource is taken away upon use
    (➔ number of resources varies)
    - **typical producer / consumer problems**
    - example: characters typed using a keyboard, blocks of data from the network

  - ***reusable* resources**: resource is given back after use
    (➔ number of resources is fixed)
    - Typically, **mutual exclusion (critical section)** or **readers/writers type of problems**
    - example: processor, memory blocks, physical entities, variables

Deadlock: analysis

Next video:

- **Deadlock detection algorithm** depends on the type of resource:
  - **Consumable:** *wait-for graph*
  - **Reusable:** *dependency graph*

Operating Systems (2INC0)

Deadlock (08)
Deadlock analysis

Dr. Geoffrey Nelissen

Courtesy of Prof. Dr. Johan Lukkien and
Dr. Tanir Ozcelebi

(also thanks to Bic & Shaw, Silberschatz, Galvin & Gagne)

IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Model for analysis: graphs

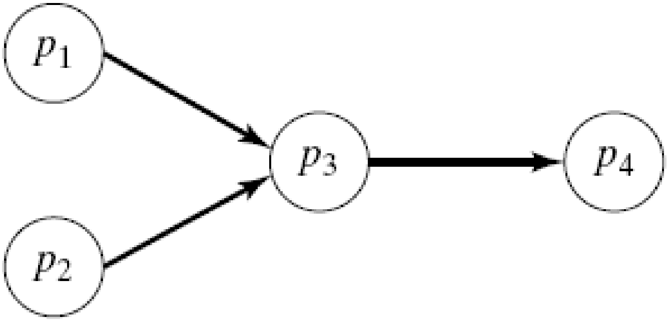
- **Analysis of deadlocks**:
  - **Consumable resources** and **condition synchronizations:**
    wait-for graph
  - **Reusable resources** and **action synchronizations:**
    dependency graph

Model for analysis: graphs

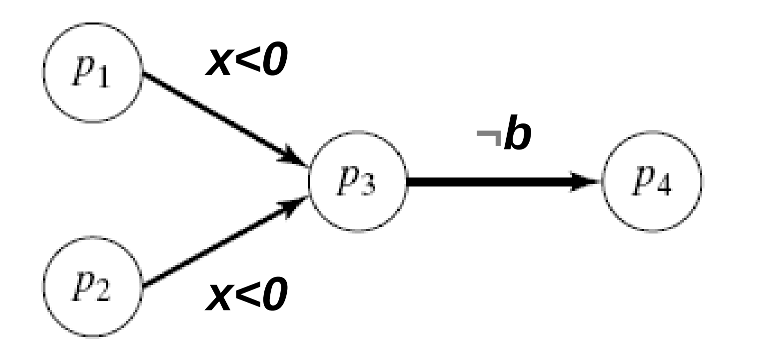
- **Analysis of deadlocks**:
  - **Consumable resources** and **condition synchronizations:**
    wait-for graph
  - **Reusable resources** and **action synchronizations:**
    dependency graph

Wait-for graph

- ***Wait-for*** **graph:**
  - **Nodes = tasks**
    - i.e., the activities, thread/process
  - **Edges =** a ***wait-for*** (i.e. *blocked-on*) relationship
    - an edge ***p1* → *p3* means that *p1* may unblock *p3*** *(textbook uses it differently)*



Wait-for graph Example

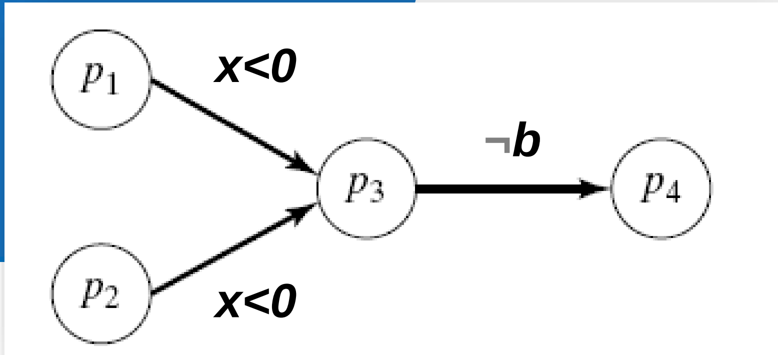


- *p1*: *... P(m); x := x+y;* ***sigall(c);*** *V(m) ...*
- *p2: ... P(m); x := a;* ***sigall(c);*** *V(m) ...*
- *p3: ... P(m);* **while** ***x<0*** **do** ***wait(m,c)*** **od*;*** *x := x-1; b := true;* ***signal(d);*** *V(m) ...*
- *p4: ... P(m);* **while not** ***b*** **do** ***wait(m,d)*** **od*;*** *... b := false; V(m) ...*

- The graph captures a *possible dynamic* situation (reachable system state),
  - We must **prove** the ***possibility of existence*** of the graph
    - e.g., if *a* is always negative, there is never a state with an arrow $p2 \rightarrow p3$
  - We **label** the edges with corresponding **blocking conditions**
    - i.e., information about the state that gives the specific blocking
    - e.g., at this state p_3 is blocked due to $x<0$ and p_4 is blocked due to $\neg b$

- Note:
  - We **leave out the dependency on the mutex *m*** since we know mutual exclusion does not add to deadlock provided that the critical sections terminate as it is the case in the example.

Wait-for graph Analysis



- **Deadlock** possible **only if** there is a **cycle** in the wait-for graph
- How to prove the absence of deadlock?
  - Proof by contradiction:
    - **Assume there is a deadlock** between the **tasks involved in a cycle**
    - **Prove that at least one task can be unblocked by a task** that is **not involved in that cycle**

Model for analysis: graphs

- **Analysis of deadlocks**:
  - **Consumable resources** and **condition synchronizations:**
    wait-for graph
  - **Reusable resources** and **action synchronizations:**
    dependency graph

Resource dependency graph

- ***Resource dependency* graph:**
  - bipartite graph with **two classes of nodes** = **tasks** and **resources**
  - **three types of edges**
    - Type1: **task** has **requested** and now waits for the **resource**
    - Type2: **resource acquired (held) by task**
    - Type3: **task may request** the **resource**

$p \longrightarrow R$: p requests R

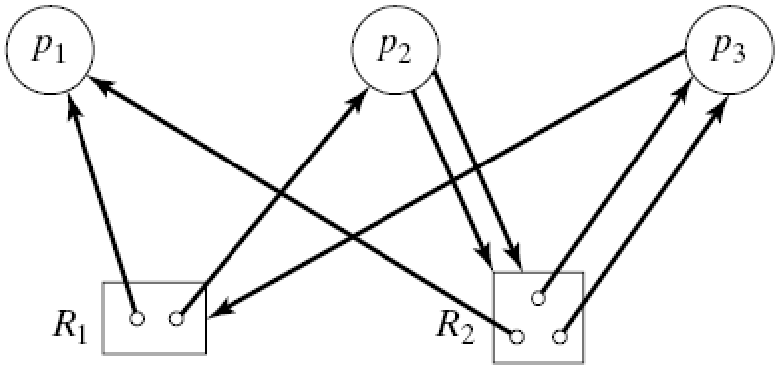
$R \longrightarrow p$: p holds R

$p \dashrightarrow R$: p *may* request R

- A **resource dependency graph** represents a **particular state of the system**
- Three type of events may change the state
  - *request* (by a task),
  - *acquire* (response to a request by the system, according to a policy),
  - *release* (by the task)

Resource dependency graph Example

- Edges
  - *p* ⟶ *R* : *p* requests *R*
  - *R* ⟶ *p* : *p* holds *R*
  - *p* - - -▸ *R* : *p may* request *R*

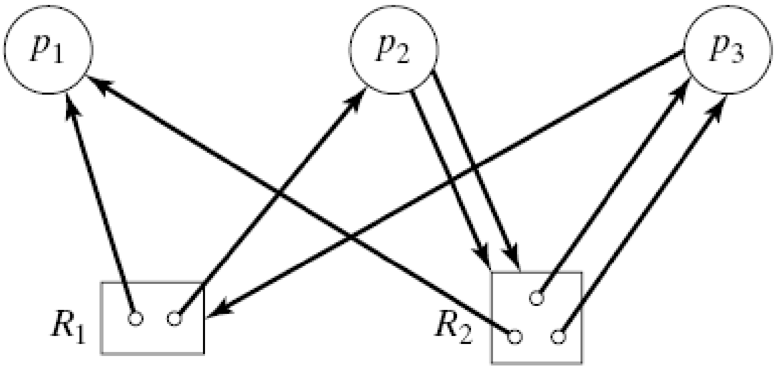


- Example (graph shown)
  - *p1* holds one of *R1 and one of R2*
  - *p2* holds one of *R1* and requests two of *R2*
  - …

- ***p* is blocked if** it has an outgoing **edge that is not directly removable**
  - i.e., for which **the requested resource is free**

Resource dependency graph Example

- Edges
  - *p* ⟶ *R* : *p* requests *R*
  - *R* ⟶ *p* : *p* holds *R*
  - *p* - - -▸ *R* : *p may* request *R*

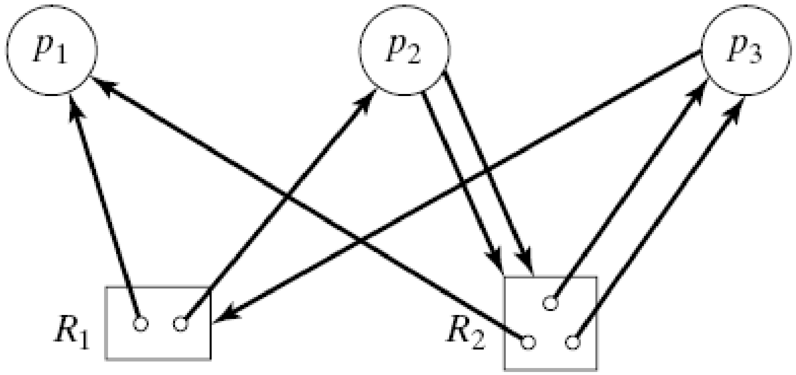


- Example (graph shown)
  - *p1* holds one of *R1 and one of R2*
  - *p2* holds one of *R1* and requests two of *R2*
  - *…*

- ***p* is blocked if** it has an outgoing **edge that is not directly removable**
  - i.e., for which **the requested resource is free**

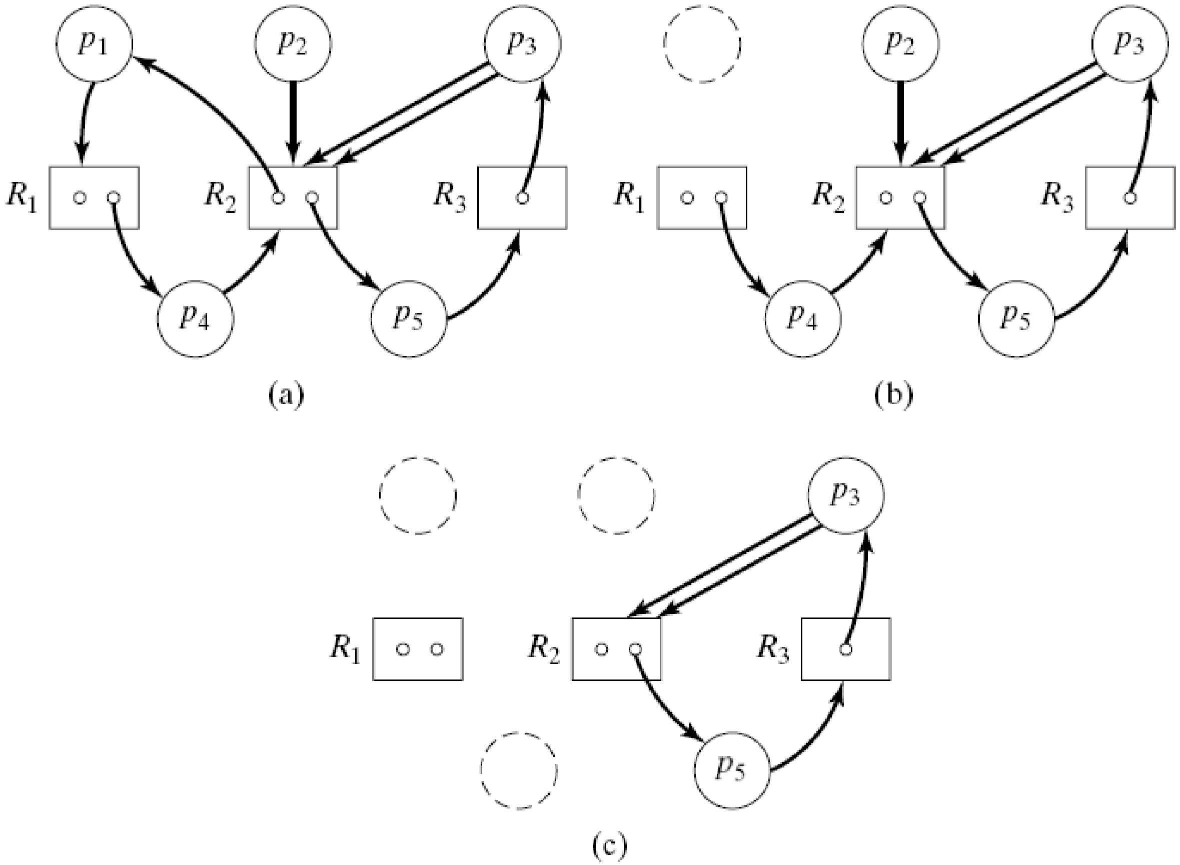
Resource dependency: reduction

- Assume that the graph represents a *stable* state**, repeatedly remove a non-blocked task** and all its incoming connections
  - Simulates the completion of that task critical section



- If there is a **remaining set** after reduction = **deadlocked**
  - Called a *knot*

Resource dependency: Reduction in progress...

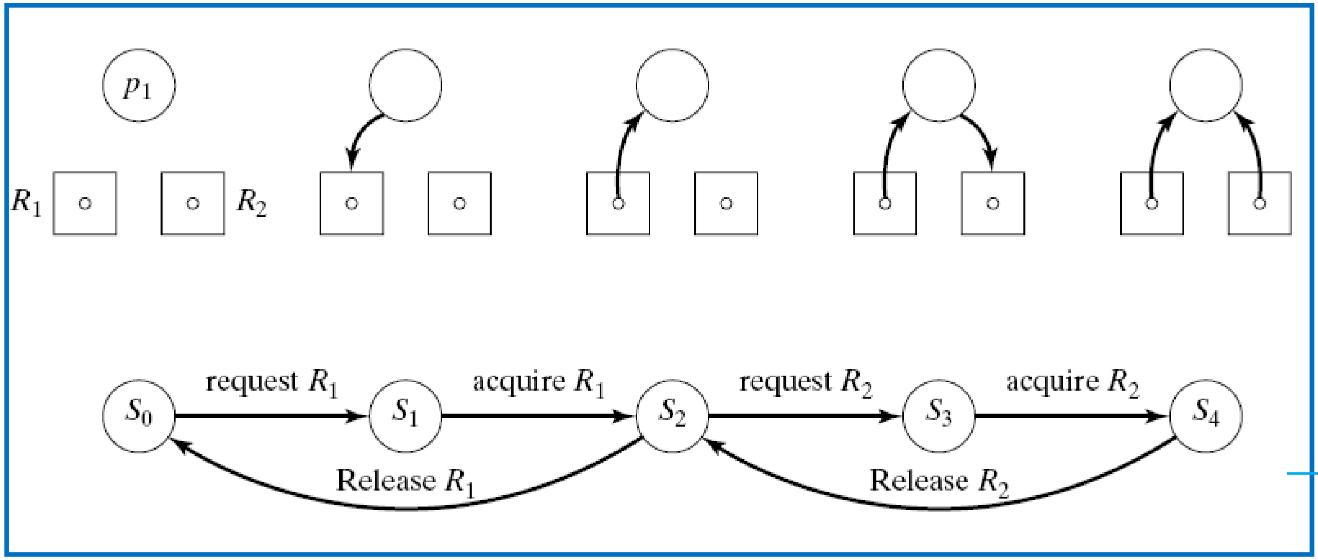


Resource dependency: Sufficient condition for deadlock

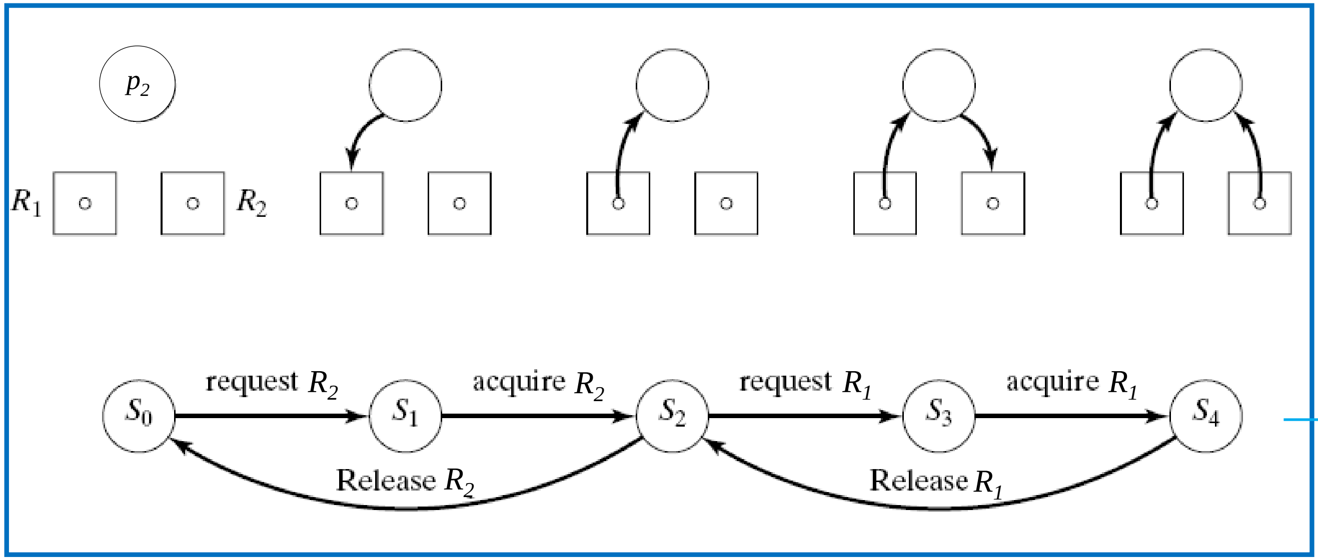
- If we have a **greedy allocation algorithm**:
  - **Allocate resources as soon as they are available**
    - could be implemented just with a semaphore (counting the resources)
  - **Show the existence of a knot** in one of the dependency graphs that can be generated **by allocating the resources arbitrarily**

- **More generally,** examine the reachable states of the **Finite State Machine** corresponding to the request/acquisition sequences

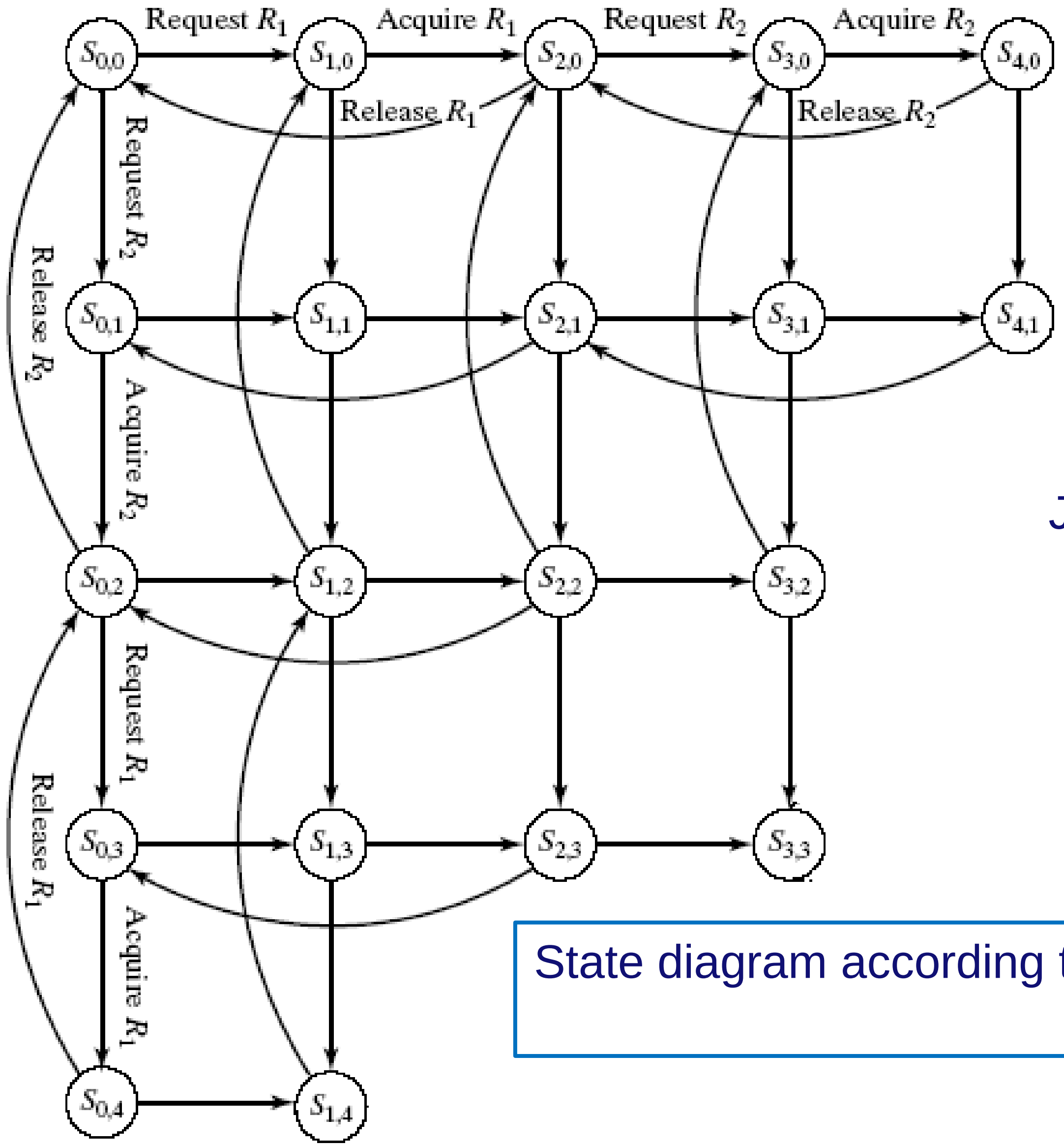
Example: state diagrams and reachable states



Finite State Machine for *p1.*

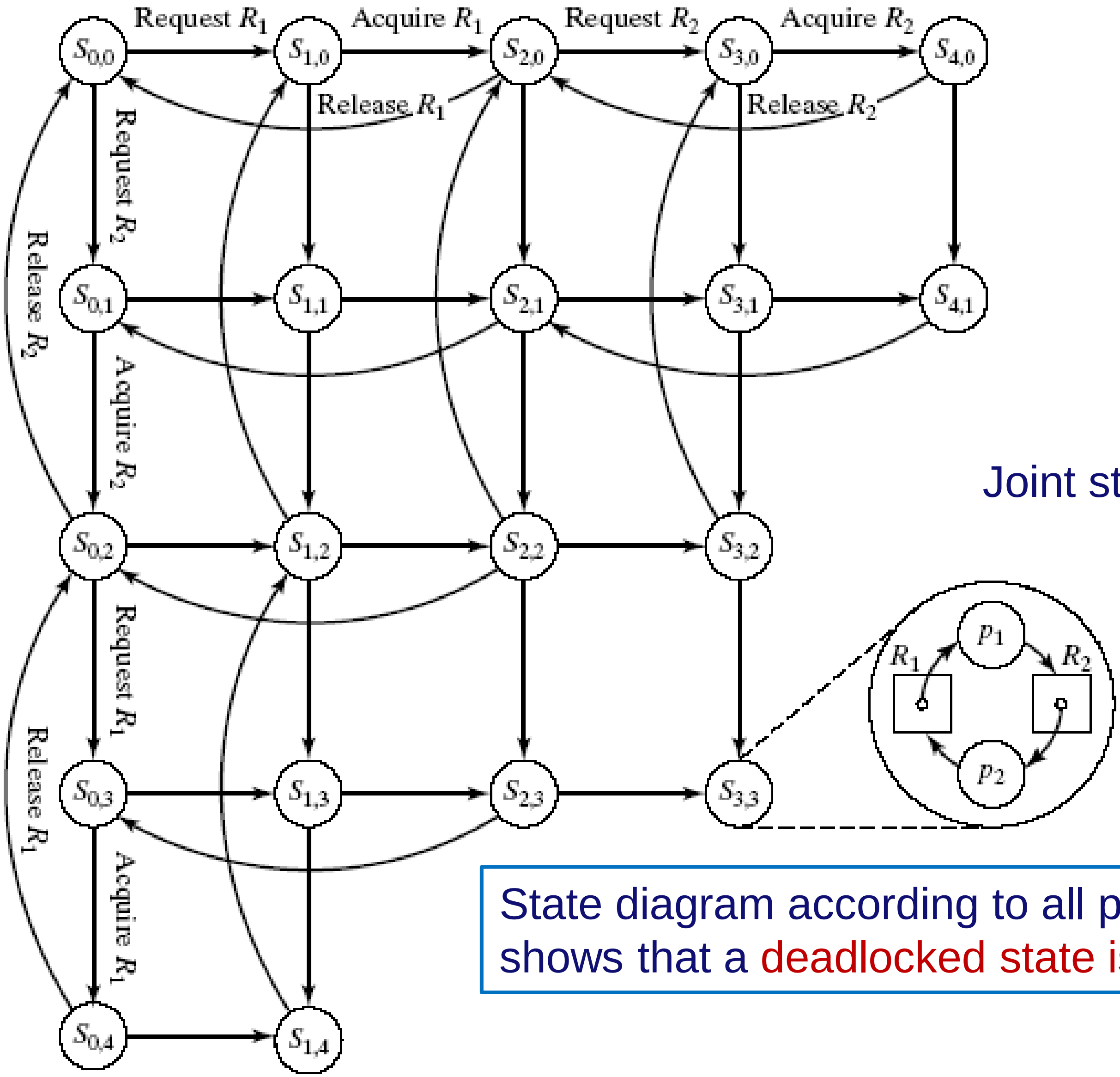


Finite State Machine for *p2.*



Joint state space (FSM for it).

State diagram according to all possible traces



Joint state space (FSM for it).

State diagram according to all possible traces shows that a deadlocked state is reachable.